TOPIC 3

THREADING UTILITIES

ATOMIC

ATOMIC VARIABLES

- Based on a branch of research focused on creating non-blocking algorithms for concurrent environments
- Algorithms exploit low-level atomic machine instructions such as compare-and-swap (CAS)

COMPARE AND SWAP

A typical CAS works on three operands:

1. The memory location on which to operate (M)
2. The existing expected value (A) of the variable
3. The new value (B) which needs to be set

Atomically set the value in **M** to **B**, but only if the existing value in **M** matches **A**, otherwise no action is taken - All as a single operation

WHAT IS DIFFERENT?

- No Threads are suspended
- They are informed that they have not completed the update
- You may have to handle the situation where a CAS operation did not succeed.
    - Retry
    - Do Nothing

COMMON ATOMIC VARIABLES

- `AtomicInteger`
- `AtomicLong`
- `AtomicBoolean`
- `AtomicReference`

COMMON METHODS IN ATOMIC VARIABLES

- `get()` - gets the value from the memory, like `volatile`
- `set()` - write a value to memory, like `volatile`
- `lazySet()` - eventually writes the value to memory, maybe reordered with subsequent relevant memory operations
- `compareAndSet()` - compare value, if maintained, write new value. returns `true` when it succeeds, else `false`

DEMO: CONCURRENTSTACK

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.atomic` package and open *ConcurrentStack.java*

ACCUMULATORS

`LongAdder`

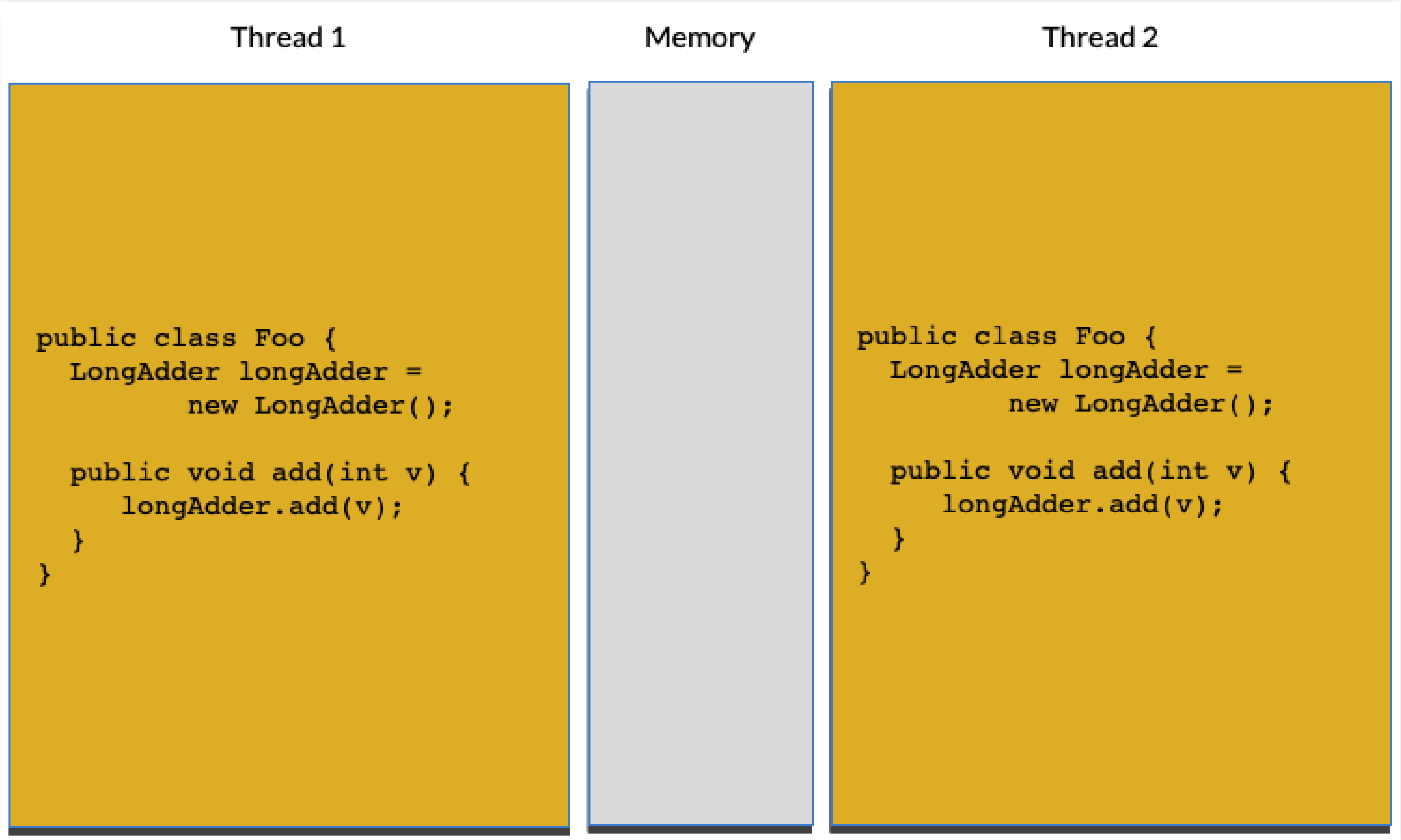
- One or more variables that together maintain an initially zero long sum.
- When updates like method `add(long)` are contended across threads, the set of variables may grow dynamically to reduce contention.
- Method `sum()` (or, equivalently, `longValue()`) returns the current total combined across the variables maintaining the sum
- Preferrable to `AtomicLong` for accumulation
- Extends `Number`

PERFORMANCE OF `LongAdder`

- Under low update contention, the two classes have similar characteristics
- But under high contention, expected throughput of this class is significantly higher
  - At the expense of higher space consumption

LONG ADDER BY EXAMPLE

In `LongAdder` each thread is maintaining in it's own count



Thread 1

```
public class Foo {
  LongAdder longAdder =
         new LongAdder();

  public void add(int v) {
     longAdder.add(v);
  }
}
```

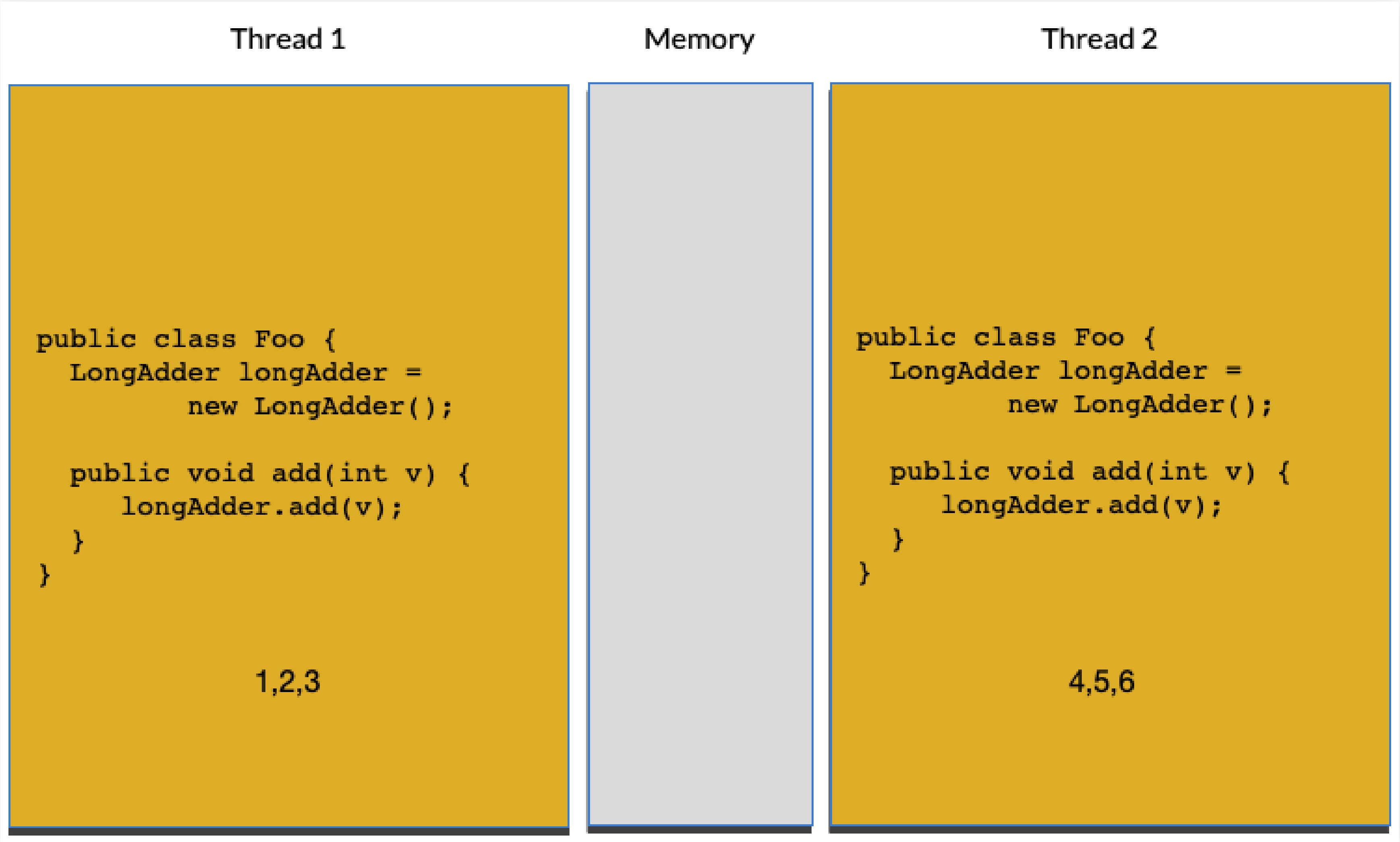
Memory

Thread 2

```
public class Foo {
  LongAdder longAdder =
         new LongAdder();

  public void add(int v) {
     longAdder.add(v);
  }
}
```

LOCALLY ADDING THE NUMBERS



Thread 1

```
public class Foo {
  LongAdder longAdder =
         new LongAdder();

  public void add(int v) {
     longAdder.add(v);
  }
}
```

1,2,3

Memory

Thread 2

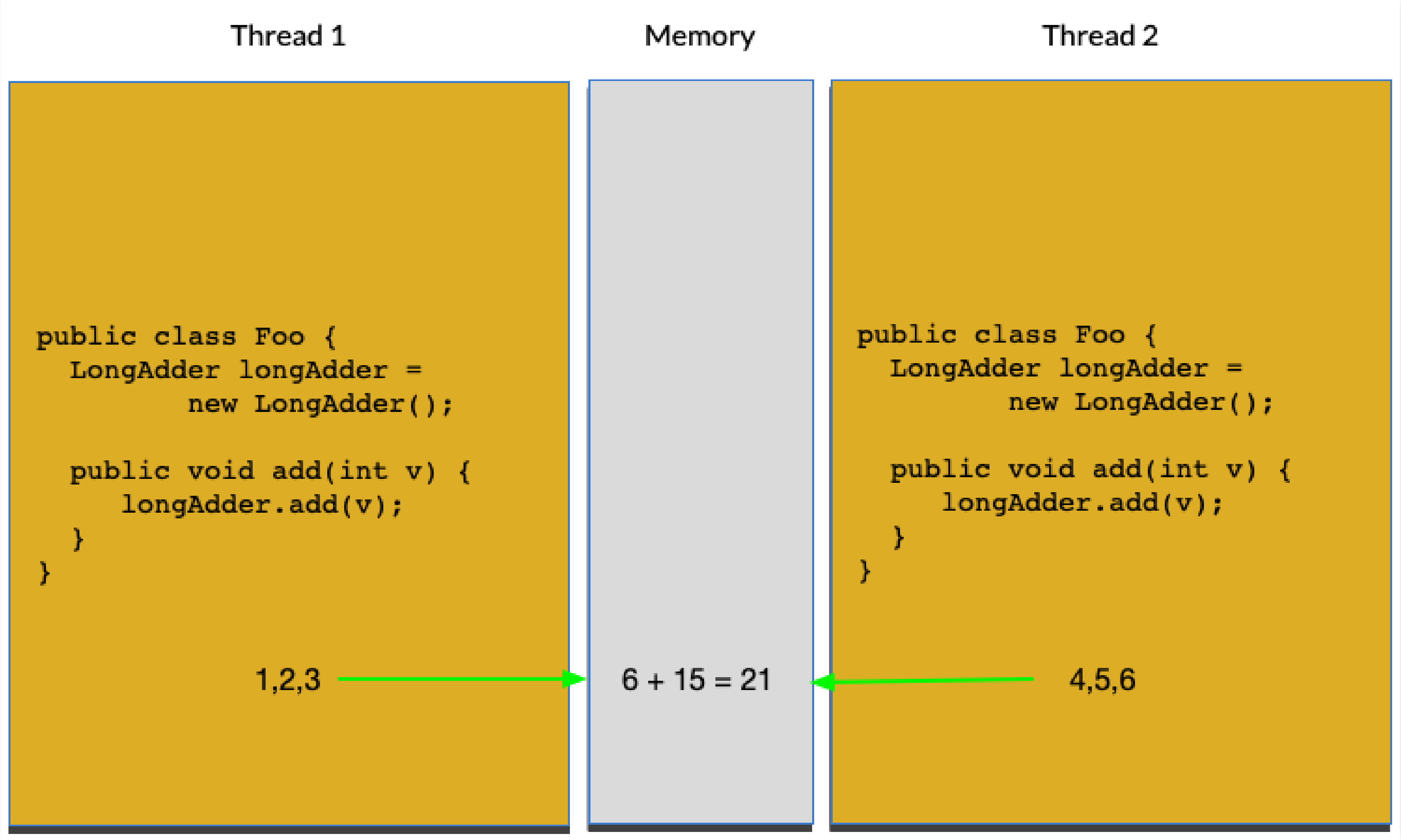
```
public class Foo {
  LongAdder longAdder =
         new LongAdder();

  public void add(int v) {
     longAdder.add(v);
  }
}
```

4,5,6

COMBINING THE NUMBER INTO MAIN MEMORY

Since Adding is Commutative, we can bring the values in together when done



```
public class Foo {
  LongAdder longAdder =
         new LongAdder();

  public void add(int v) {
     longAdder.add(v);
  }
}
```

```
public class Foo {
  LongAdder longAdder =
         new LongAdder();

  public void add(int v) {
     longAdder.add(v);
  }
}
```

`DoubleAdder`

- Same as `LongAdder` but for `Double` floating numbers

`LongAccumulator`

- `LongAccumulator` is a General Form of the `LongAdder`, thread safe, where you can apply your own function
- Similar to `reduce` in functional programming
- Pass each number that is required to accumulate with the `accumulate` method

`DoubleAccumulator`

- Same as `LongAccumulator` but for `Double` floating numbers

DEMO: ADDERS AND ACCUMULATORS

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.accumulators` package and the files in the package

CONCURRENT COLLECTIONS

`BlockingQueue`

- `Queue` that additionally supports operations that wait for the queue to become non-empty when retrieving an element
- Wait for space to become available in the queue when storing an element
- Subtypes include:
  - `ArrayBlockingQueue`
  - `DelayQueue`
  - `LinkedBlockingDeque`
  - `LinkedBlockingQueue`
  - `LinkedTransferQueue`
  - `PriorityBlockingQueue`
  - `SynchronousQueue`

WHAT IF THE OPERATION CANNOT BE SATISFIED?

+	Throws Exception	Special Value (`null` or `false`)	Blocks	Times out
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(e, time, unit)`
Examine	`element()`	`peek()`	*n/a*	*n/a*

`ConcurrentMap`

- A `Map` providing thread safety and atomicity guarantees, since regular `Map` are not Thread-safe
- Actions in a thread prior to placing an object into a `ConcurrentMap` as a key or value happen-before actions subsequent to the access or removal of that object from the `ConcurrentMap` in another thread.
- Is an extension of `Map`
- Subtypes include:
  - `ConcurrentHashMap`
  - `ConcurrentSkipListMap`

`ConcurrentNavigableMap`

- An interface that has navigable methods for the `Map`
- Backed by `SortedMap` where keys are sorted
- Includes
  - `subMap` - Returns a portion of the `Map` given by keys
  - `headMap` - Returns a view of the portion of this map whose keys are strictly less than a key
  - `tailMap` - Returns a view of the portion of this map whose keys are strictly greater than a key
  - `descendingMap` - Returns a reverse order view of the mappings contained in this map.
  - `descendingKeySet()` - Returns a reverse order NavigableSet view of the keys contained in this map.
- Subtypes include: `ConcurrentSkipListMap`

COUNTDOWN LATCH

COUNTDOWN LATCH DESCRIBED

- You want a thread to wait until one or more events have occurred
- Initially created with a count of the number of events that must occur before the latch is release
- Each time an event happens, the count is decremented
- A form of `Abstract-QueuedSynchronizer`

HOW IT IS USED?

1. Instantiate the `CountDownLatch` with the same value for the counter as a number of threads we want to work across.
2. Call `await()` on the `CountDownLatch` instance to hold
3. Call `countdown()` on the `CountDownLatch` instance to `countdown()`
4. At zero, all `Thread` that are waiting will now continue.

DEMO: COUNTDOWN LATCH

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.concurrency.countdownlatch` package and open all three files in the package

CYCLIC BARRIER

WHAT IS IT?

- Allows a fixed number of parties to rendezvous repeatedly at a barrier point
- Useful in parallel iterative algorithms that break down a problem into a fixed number of independent subproblems
- Threads call `await` when they reach the barrier point
- `await` blocks until all the threads have reached the barrier point
- If all threads meet at the barrier point, the barrier has been successfully passed
  - All Threads are released
  - The Barrier is reset

WHAT HAPPENS IF THERE ARE PROBLEMS?

- If a call to `await` times out or a thread blocked in await is interrupted, then the barrier is considered broken and all outstanding calls to await terminate with `BrokenBarrierException`
- If the barrier is successfully passed, await returns a unique arrival `index` for each thread, which can be used to “elect” a leader that takes some special action in the next iteration.

ALTERNATE CALLS TO `CyclicBarrier`

- `CyclicBarrier` also lets you pass a barrier "action" to the constructor
- Action is a `Runnable` that is executed as a subthread before the blocked threads are released.

DEMO: CYCLIC BARRIERS

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.concurrent.cyclicbarrier` package and open the two classes.

PHASER

WHAT IS IT?

- A reusable synchronization barrier, that moves in phases
- Similar in functionality to `CyclicBarrier` and `CountDownLatch` but supporting more flexible usage.
- Tasks may be registered at any time with `register` and `bulkRegister`
- Arrivals are when a thread has reached the barrier and awaits until others have reached that same barrier

REGISTRATION

- Adds a new unarrived party to this phaser.
- If an ongoing invocation of `onAdvance` is in progress, this method may await its completion before returning
- Returns the arrival phase number to which this registration applied.
- If the return value is negative, then this phaser has terminated, in which case registration has no effect.

ARRIVAL

- Thread signals it has arrived with `arriveAndAwaitAdvance()` and block until others in the party have reached the barrier
- When the number of arrived parties is equal to the number of registered parties, the execution of the program will continue
- Phase number will increase, and we can obtain that with `getPhase()`

DEREGISTERING

- When done, we can call `arriveAndDeregister()` signaling that we are no longer a member of the party

DEMO: PHASERS

In the *java-targeted-topics/src/main/java* directory, navigate to the `com.evolutionnext.concurrent.phaser` package and open the two classes.